Handling Multiple Tasks

With a complex system:

- Often have many different tasks to be performed
- These tasks can have different timing requirements:
  - How often they must be performed
  - How quickly they must respond to an event

The Multi-Tasking Abstraction

This abstraction is key to building complex systems

- We can construct our system as a set of compartmentalized modules
- Each module can be implemented and tested separately
- It is easy to “mix and match” modules depending on the application

The Multi-Tasking Abstraction

This abstraction is key to building complex systems

- Each process has the “illusion” of owning the processor all of the time
- Allows for efficient use of the CPU and other system resources

Multi-tasking

- At any one time, a single process is in control of (or "owns") the processor
  - We refer to this process as being in a **running state**
- All other processes are either:
  - In a **waiting state**: waiting on some external or internal event
  - In a **ready state**: ready to execute when the processor is free

An Example: USC AFV

Sensors:

- Downward-oriented sonars: height and attitude
- Compass: yaw direction
- Rotor encoder: rotational velocity
- Downward-looking camera: position on field

An Example: USC AFV

Actuators:

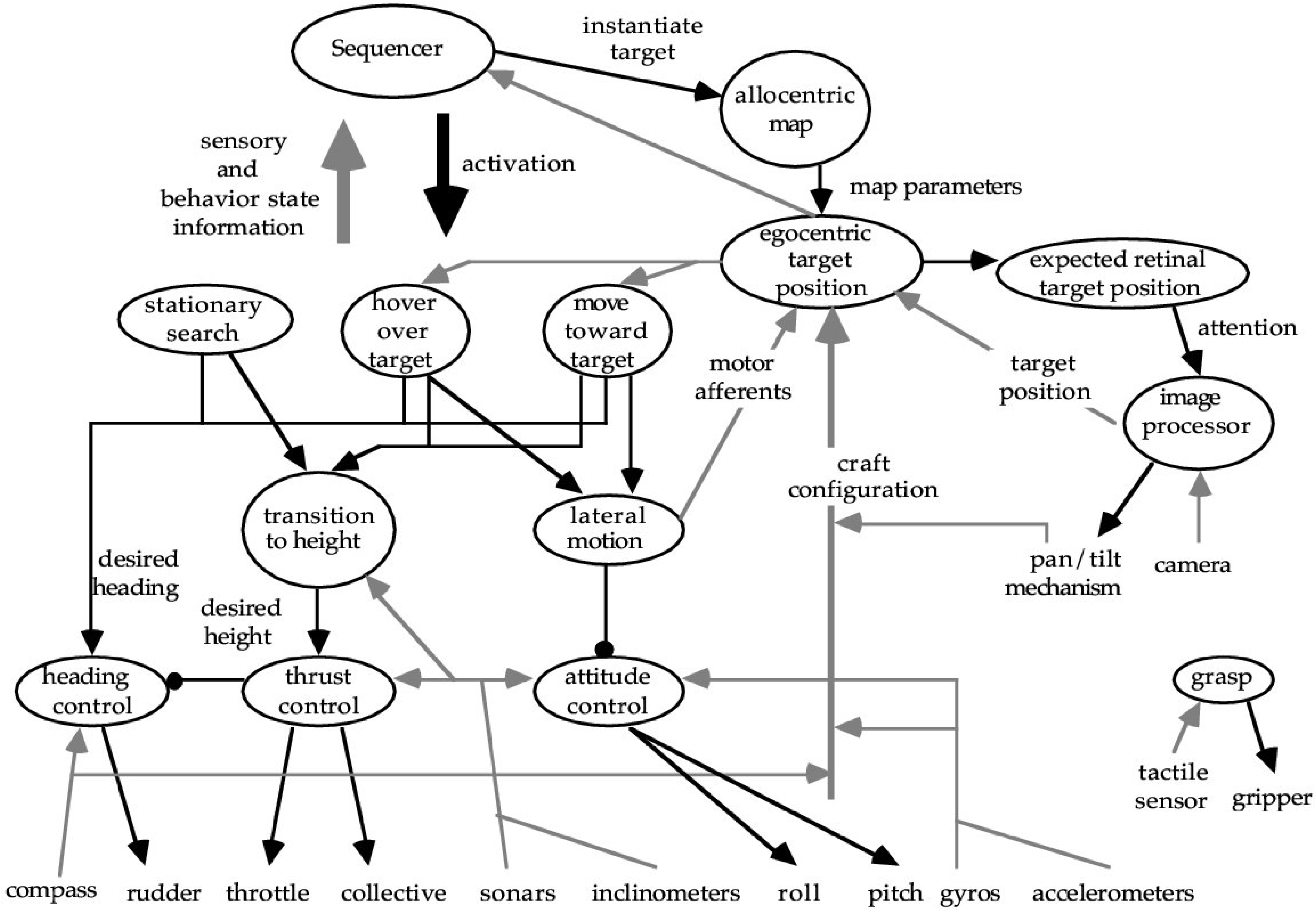
- Rotor collective
- Rotor torque
- Rotor pitch
- Rotor yaw
- Rudder

An Example: USC AFV

Tasks include:

- Thrust control
- Attitude control
- Heading control
- Move to height
- Search for target
- Hover over target
- Planner

AFV Process Architecture



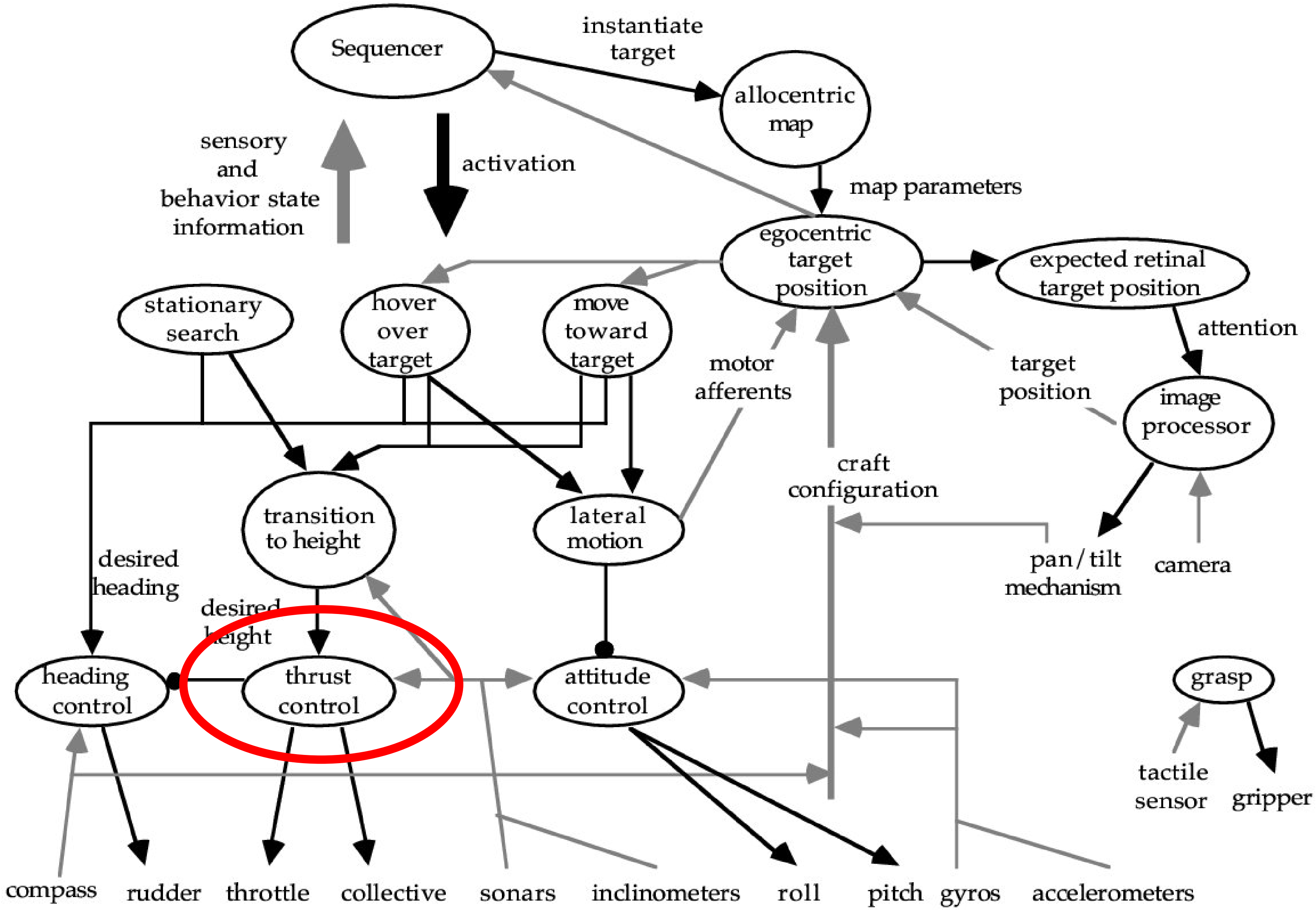
Multi-Tasking Components

- Process control block (PCB): data structure that describes the process
- Scheduling: deciding which process to execute now
- Inter-task communication: moving data between processes
- Synchronization: mechanism for safely coordinating the actions of two or more processes

Operations on a Process

- Creation (fork/exec/spawn)
- Suspend: stop a process temporarily
- Resume: undo the suspend
- Destroy: stop executing the process and deallocate its memory

A Process Example



An Example: Altitude Control Process

"BURTE" kernel

```
void altitude_servo_loop()
{
   set_schedule_interval(10);  // 10ms
   while(1)
        {
              collective = Kp * (height_desired – height)
                           - Kv * height_velocity;
              set_collective(collective);

              next_interval(); // Wait for the next control
                                  //  cycle
        };
};
```

An Example: Starting the Process

```
main()
{

            :
  pid = create(altitude_servo_loop, 10, 3000);
  start(pid);
            :
};
```

An Example:
Starting the Process

```
main()
{

             :
  pid = create(altitude_servo_loop, 10, 3000);
  start(pid);
             :
};
```

Name of function

An Example: Starting the Process

```
main()
{

            :
  pid = create(altitude_servo_loop, 10, 3000);
  start(pid);
            :
};
```

Priority

An Example:
Starting the Process

```
main()
{

               :
  pid = create(altitude_servo_loop, 10, 3000);
  start(pid);
               :
};
```

Size of stack

An Example:
Starting the Process

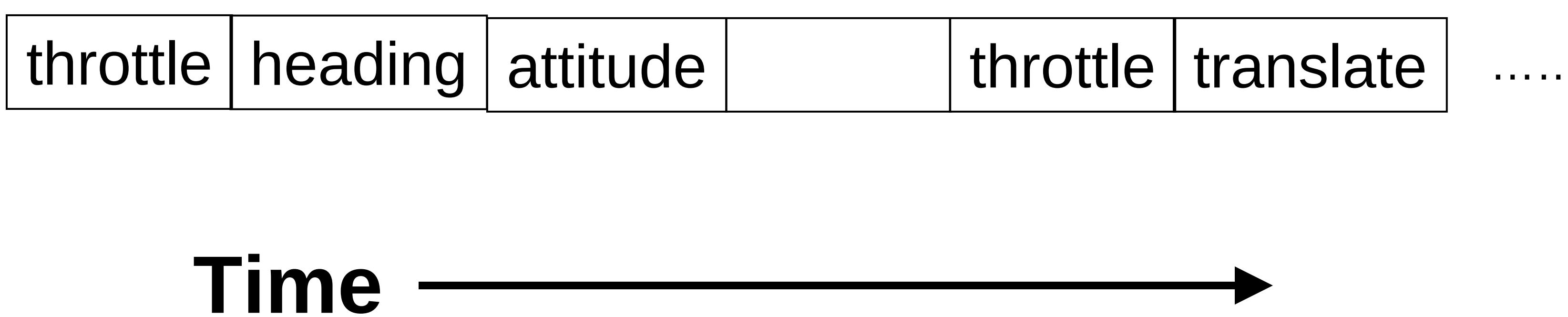
```
main()
{

                :
   pid = create(altitude_servo_loop, 10, 3000);
   start(pid);
                :
};
```

Start the process

Selecting a Process to Execute

Only one process may occupy the processor at any one time…



Selecting a Process to Execute

A **scheduler** is responsible for selecting the next process

- How might we do this?

Scheduling Policies

Only processes in the **ready state** may be selected

- Round robin: rotate between the different processes
- Priority-based: select the highest-priority process that is ready to execute
- Shortest-process-first: select the one that will use the processor for the shortest period of time

- Preemption: interrupt an executing process

Evaluating Scheduling Policies

Metrics for evaluation include:

- **Response time**: time for a process to move from ready to running
- **Turn-around time**: time for a process to move from ready to running and then to leave running
- **Throughput**: number of processes that can be executed in a given period of time
- **Overhead**: the amount of time required by the operating system to perform scheduling

Evaluating Scheduling Policies

Other key concepts:

- **Fairness**: all processes get some access to the processor (and other resources)
- **Starvation**: a process never gets access to the processor (because other processes are occupying it)

Round Robin Scheduling

- Queue: an ordered list of processes that are in a **ready** state
- Selecting the next processes: remove the process from the beginning of the queue
- Any new processes: add to the end of the queue

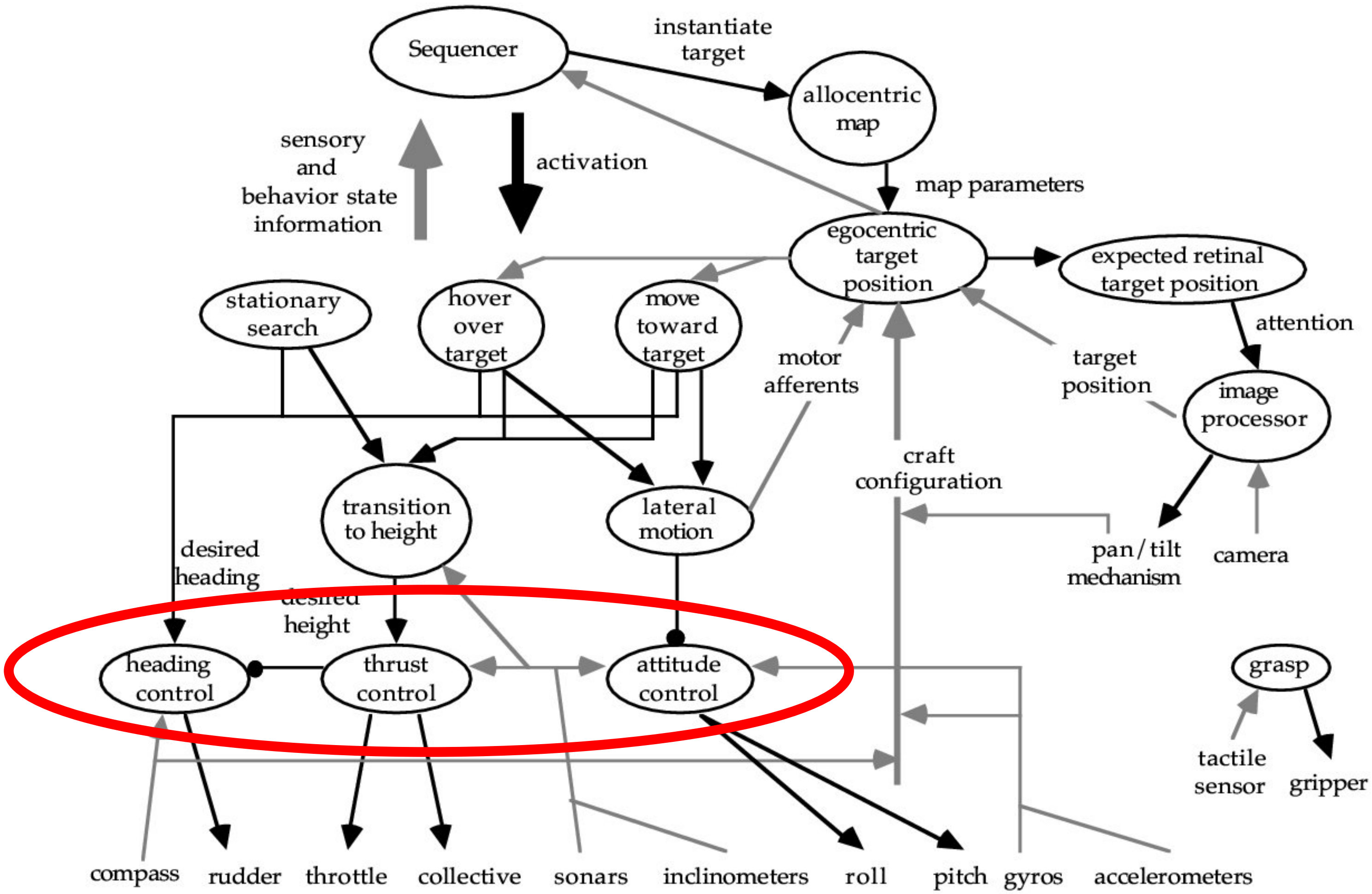
Priority-Based Scheduling

- Each process is assigned an integer **priority**
- Selecting the next process: of all the processes that are **ready**, pick the one with the highest priority

Hybrid Scheduler Example

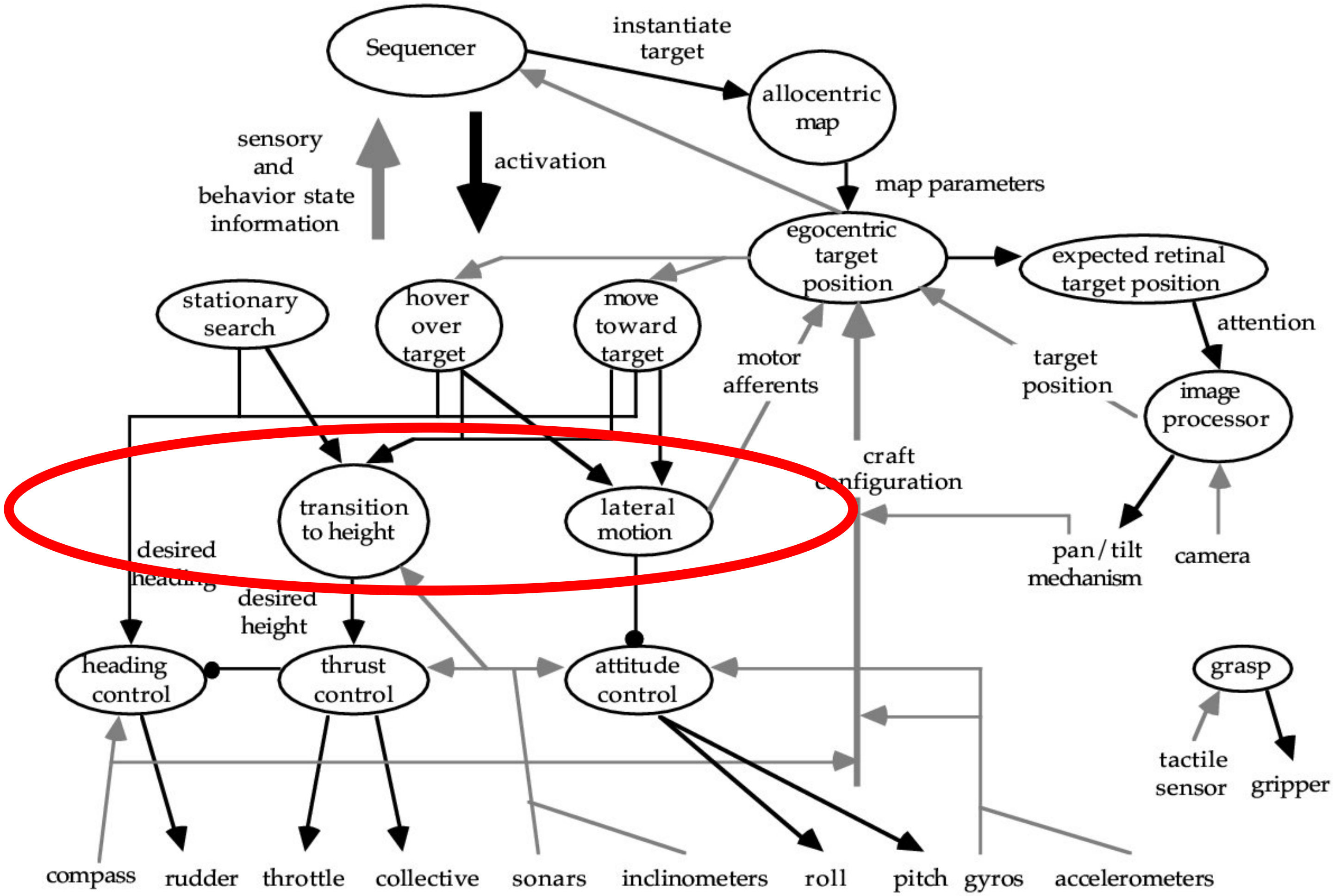
- Have a queue for each distinct priority level
- Use round robin for the highest priority queue
- If there are no processes to execute, then perform round robin between the processes in the next queue
- Repeat

Heli Example



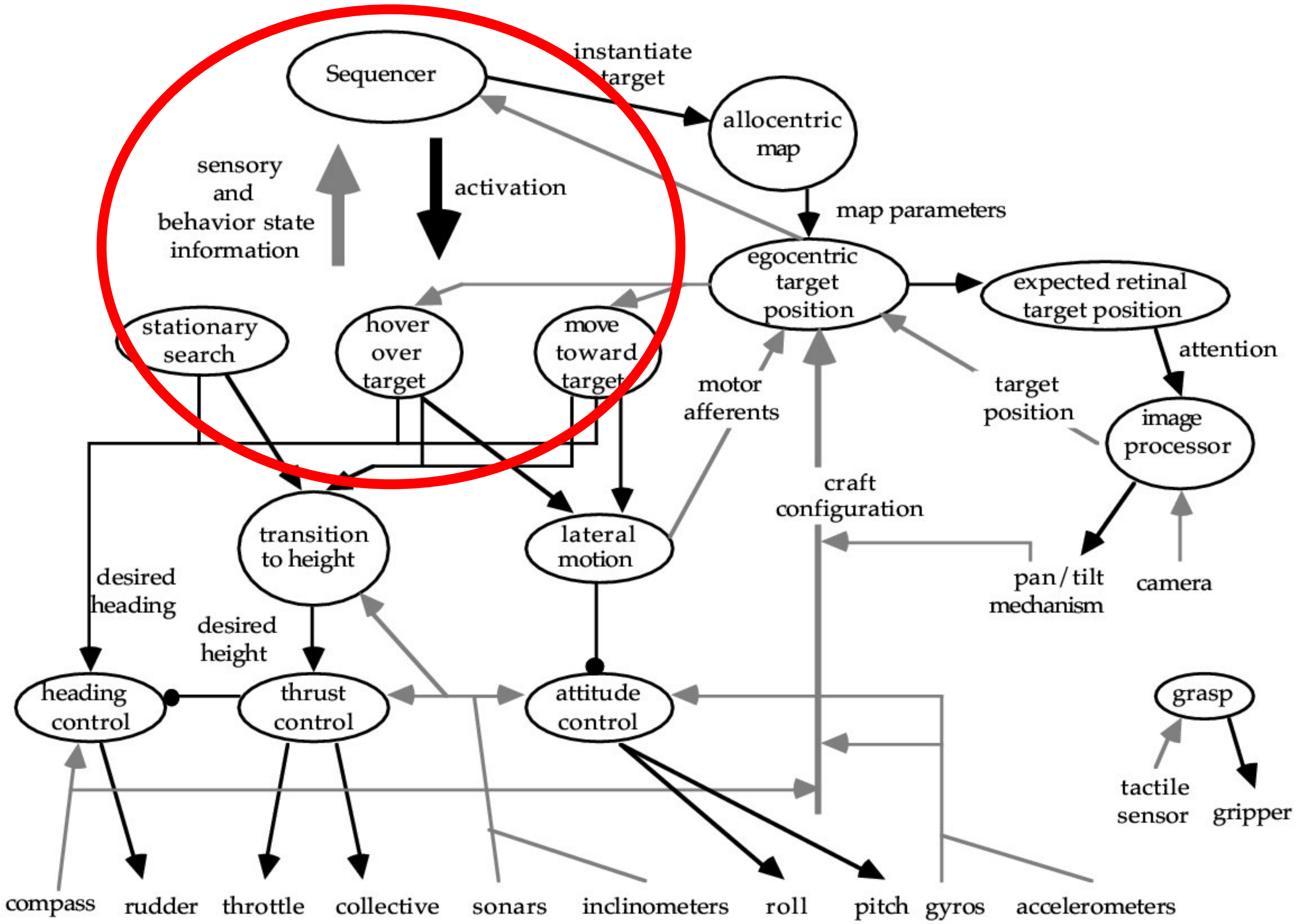
Processes with strict timing requirements are the highest priority processes

Heli Example



Many processes operate on timescales of seconds

Heli Example



Other processes operate at timescales of 10s of seconds

Non-Preemptive Scheduling

A process voluntarily gives up the processor

- This works if we are careful in our implementation
- But – we can have problems if a process does not “play nice”

Preemptive Scheduling

A process can be forced off the processor by the operating system

- Typically, a process is given a fixed-duration **timeslice** in which to execute
- If the process does not give up the processor within this time:
  - A different process is given the processor
  - The process is returned to the **ready** state

Hybrid Scheduler II

Combine preemption and priority-based scheduling

Hybrid Scheduler II

Combine preemption and priority-based scheduling ("priority preemptive scheduling")

- A process can be preempted at any time by a higher-priority process

Scheduling Regular Tasks

In many control systems, tasks (processes) must be executed at a regular frequency

- How can we be sure that all tasks can be performed?

Rate Monotonic Scheduling

- Preemptive scheduling
- Process priority is proportional to execution frequency

Rate Monotonic Scheduling

N tasks:

- T_i = the period between executions of task i
- E_i = worst case execution time
- So: E_i/T_i = the fraction of processor time required by task i

Requirement: a process must complete its i^{th} execution before i+1 enters the ready queue

RMS Theorem

A set of processes is schedulable if:

$$\sum_i \frac{E_i}{T_i} \leq n\left(2^{1/n} - 1\right)$$

An Example Scheduling Problem

	T_i	E_i
Process 1	100 ms	30 ms
Process 2	250 ms	40 ms
Process 3	1 s	60 ms

- All start in the ready queue at time 0
- Process 1 is first in the queue (2 is the 2nd)
- Round Robin scheduling, non-preemptive

What is the sequence of execution?

An Example Scheduling Problem

	T_i	E_i
Process 1	100 ms	30 ms
Process 2	250 ms	40 ms
Process 3	1 s	60 ms

- All start in the ready queue at time 0
- Rate Monotonic scheduling
- Does it meet the criterion?
- What is the sequence of execution?

Example II

	T_i	E_i
Process 1	50 ms	25 ms
Process 2	100 ms	40 ms

Scheduling algorithm: priority with preemption

- Assume Process 2 has highest priority

Scheduling algorithm: RMS

- What choice would **Rate Monotonic Scheduling** make about priority?
- Does RMS say that these processes are necessarily schedulable?

RMS Scheduling

What did we learn?

- Priority matters!
- The Rate Monotonic Scheduling constraint is a **sufficient** condition for schedulability - but not a **necessary** one

Example III

	T_i	E_i
Process 1	50 ms	25 ms
Process 2	75 ms	30 ms

What is the total processor utilization?

Example III

	T_i	E_i
Process 1	50 ms	25 ms
Process 2	75 ms	30 ms

What is the total processor utilization?

90%

Do we pass the RMS constraint?

Rate Monotonic Scheduling

	T_i	E_i
Process 1	50 ms	25 ms
Process 2	75 ms	30 ms

Do we pass the RMS constraint?

NO

What is the schedule anyway?

RMS Scheduling

What did we learn?

- CPU underutilized does not imply that a schedule exists